Evaluating Improper Integrals with Cubic Spline Interpolation

Abraham Flores
Advisor: Dr. Mark Caprio
University of Notre Dame

June 20 2016
Edited: July 14 2016

1 Introduction

This pdf will serve as a guide to the C++ routines and the process we used to evaluate these integrals. The starting point of this project was to use two computational methods to evaluate integrals from zero to infinity. First we would analytically change the bounds of the integral to be zero to one. From there if we could generate the integrand we could then use the GNU Scientific Library (GSL) to generate cubic splines of the integrand. Making the integral into trivial power rule integration.

1.1 Requirements

In order to make use of the routines I have created you must have the *GSL* built and ready to use. If you have windows, use MingGW and built GSL with msys correctly (putting the library into the MinGW Include and LIB folder) than the makefiles should* work. If not see Additional comments for compile instructions.

In order to use the cs_basis files you will need *Eigen*.

Link to DropBox containing files

2 Process

This section will describe the mathematical methods and coding details necessary to have a good understanding of where the core pieces of code comes from.

2.1 Change of Variable

If we did not translate the integral to finite bounds than we would have to truncate our integral and risk losing information. Using the process outlined in *Computational Physics* our change of variable to z becomes

$$z = \frac{r}{r+1} \quad \textit{or equivalently} \quad r = \frac{z}{1-z} \tag{1}$$

$$\textit{Then } dr = \frac{dz}{(1-z)^2} \tag{2}$$

$$\textit{Yielding } \int_0^\infty f(r)dr = \int_0^1 \frac{1}{(1-z)^2} f(\frac{z}{1-z})dz \tag{3}$$

Now we have our integrand on numerical boundaries. Within given_func() the new variable z is passed in, so we must solve for r using (1) from there we easily the integrand, by declaring dr as (2) then coding our function and returning the product of the two. Once we can do this we are ready to generate the splines.

2.2 Cubic Spline Interpolation

The heavy mathematics of Cubic Spline Interpolation is taken care of by the GSL however here is the outline. We have an array of data points x and y. We can break up the data points into groups, say groups of four because we need exactly four to match a cubic function to four data points. That may sound easy but we will need to link the endpoints of each group. Meaning each generated cubic will depend on the last generating a large system of equations. Which can be readily solved as a matrix problem. Here are the Restrictions put in place for Cubic Spline Interpolation by the GSL:

-Take vector or array of data points separate into groups of four
-Approximate each group with a cubic function
-Model given function with piecewise cubic
-Match the first and second derivative at each data point
-first and last endpoint have their second derivative set to zero

2.3 GSL Routines and Types

GSL Interpolation types:

gsl_interp_accel: This function returns a pointer to an accelerator object, which is a kind of iterator for interpolation lookups. It tracks the state of lookups, thus allowing for application of various acceleration strategies.

gsl_interp_cspline: Cubic spline with natural boundary conditions. The resulting curve is piecewise cubic on each interval, with matching first and second derivatives at the supplied data-points. The second derivative is chosen to be zero at the first point and last point.

gsl_spline:Returns a pointer to a newly allocated interpolation object of type t for n size data-points

GSL Interpolation Functions:

Using the higher interface functions we can just store our spline values within the spline data structure.

gsl_spline_init(gsl spline * spline, const double xa[ ], const double ya[ ], size t size): This function initializes the interpolation object interp for the data (xa,ya) where xa and ya are arrays of size size. The interpolation object The xa data array is always assumed to be strictly ordered, with increasing x values; the behavior for other arrangements is not defined.

gsl_spline_eval_integ(const gsl spline * spline, double a,double b, gsl_interp_accel * acc): Evaluates the given data points

These next routines just free the data structures and delete their memory.

gsl_spline_free (gsl_spline):

gsl_interp_acce_free (gsl_interp_accel * acc);

3 Routines

This section will define the necessary arguments, returns and the use of each routine available. General Spline makes use of *Function pointers* Here is example showing one way to define a function pointer.

```
double foo(double x){return x*x*exp(-x);}

int main ()
{
    int n=100000;
    std::pair<double,double> bounds = {0,1000};
    bool cov = false;

    double (*my_func)(double) = foo;

    std::cout<<"Integral Value: "<<integrate(n,bounds,cov,my_func);

  return 0;
}
```

Figure 1: General Spline Example

Here foo is my function that I wish to integrate from 0 to 1000 and my_func is my function pointer.

3.1 spline.h

given_func(double z, bool cov, double(*given)(double))

Arguments:

z: the variable of integration

cov: Switch for Change of Variable

given: a function pointer that points to your function

Return: (double) given_func calculates the value of your function at point z.

Usage: it is called in get_data to generate the individual data points.

get_data(double *x,double *y, int n, pair$<double, double>$ bounds, bool cov, double(*given)(double))

Arguments:

x: Array of Doubles taken by reference, x-axis of data points

y: Array of Doubles taken by reference, y-axis of data points

n: Integer number of data points. This is also the number of points in each array

bounds: Stores the bounds of integration

cov: Switch for Change of Variable

given: a function pointer that points to your function

Return: Void

Usage: It passes the arrays by reference and passes given and cov to given_func to calculate each data point and stores them in the arrays.

spline_eval(double *x, double *y, int n)

Arguments:

x: Array of Doubles taken by reference, x-axis of data points

y: Array of Doubles taken by reference, y-axis of data points

n: Integer number of data points. This is also the number of points in each array

Return: (double) Using the GSL routines it evaluates and returns the given integral

Usage: Takes two arrays of matching x-y data points and approximates the integral over the x-range

integrate(int n, pair$<double, double>$ bounds, bool cov, double(*given)(double))

Arguments:

x: Array of Doubles taken by reference, x-axis of data points

y: Array of Doubles taken by reference, y-axis of data points

n: Integer number of data points. This is also the number of points in each array

bounds: Stores the bounds of integration

cov: Switch for Change of Variable

given: a function pointer that points to your function

Return: (double) Value of your integral

Usage: evaluates your integrand on your bounds

3.2 wavefunctions.h

Class that Generates a Radial WaveFunction Data Structure. It stores the relevant data members and has some useful member functions.

Prviate Data Members

int n_ : n quantum number, default = 1

Accesed with WF.n()

int l_ : l quantum number, default = 0

Accesed with WF.l()

double b_ : Length parameter, default = 1

Accesed with WF.b()

string basis_ : Basis of the wavefunction, default = "hc"

Accesed with WF.basis()

Allowed Basis:

Harmonic Oscillator in Coordinate = "hc"

Harmonic Oscillator in Momentum = "hm"

Laguerre in Coordinate = "lc"

Laguerre in Momentum = "lm"
Constructor: WaveFunction name = WaveFunction(n,l,b,basis)

MEMBER FUNCTIONS:
int n(): returns data member n_
int l(): returns data member l_
double b(): returns data member b_
string basis(): returns data member basis_
vector$<$ *double* $>$ **find_roots()**
Uses Gauss Quadrature to determine the roots of orthogonal polynomials. See references for process.
Returns the roots of the function sorted from min to max in a vector
vector$<$ *pair* $<$ *double*, *double* $>>$ **make_bounds(WaveFunction wf)**
Calls find_roots to determine the roots of their overlap. Then makes pairs:
$[(0 -> first), (first -> middle)..., (middle -> last), (last -> \infty)]$
Returns a vector of pairs that can be used as bounds of integration. This Method does not help CSI, So I do not use it. However other Integration methods could probably make use of this function.

3.3 cs_spline.h

The purpose of spline_cs is to evaluate the inner product of Radial Wave Functions.
cs_basis(double z ,WaveFunction wf_1, WaveFunction wf_2)
Arguments:
z: The variable of integration
wf_1: WaveFunction 1 that holds its Data Members
wf_2: WaveFunction 2 that holds its Data Members
Return: (double) $\psi_1(r) \times \psi_2(r) \times dr$
Usage: Evaluates the integrand of the inner-product at point z

get_data(double *x, double *y, int n, WaveFunction wf_1, WaveFunction wf_2)
Arguments:
x: Array of Doubles taken by reference, x-axis of data points
y: Array of Doubles taken by reference, y-axis of data points
n: Integer number of data points. This is the number of points in each array
wf_1: WaveFunction 1 that holds its Data Members
wf_2: WaveFunction 2 that holds its Data Members
Return: Void
Usage: Generates the arrays to be used in spline_eval()

spline_eval(double *x, double *y, int n)
Same as spline_eval in spline.h

integrate(int n,WaveFunction wf_1, WaveFunction wf_2)
Arguments:
n: Integer number of data points. This is also the number of points in each array
wf_1: WaveFunction 1 that holds its Data Members
wf_2: WaveFunction 2 that holds its Data Members
Return: Calculates the overlap of $\psi_1(r)$ and $\psi_2(r)$
Usage: Brings all the Routines Together for easy outside use.

3.4 cs_basis.h

factorial(int n)
Arguments:
Integer n
Return: (double) n!
Usage: Calculates the factorial of n

laguerre_polynomial(int n, int k, double x)
DO NOT USE THIS ROUTINE: Not accurate enough
Arguments:
n: Order of Laguerre polynomial
k: Order of the generalized Laguerre polynomial
x: point to evaluate the Laguerre polynomial at
Return: (double) Evaluates L_n^k(x)
Usage: Needed for the Coordinate space Radial wavefunctions in the Laguerre basis

jacobi_polynomial(int n, double a, double b, double x)
Arguments:
n: Order of Jacobi polynomial
a: parameter of Jacobi polynomials
b: parameter of Jacobi polynomials
x: point to evaluate the Jacobi polynomial at
Return: (double) Evaluates $P_n^{(a,b)}$(x)
Usage: Needed for the Momentum space Radial wavefunctions in the Laguerre basis

The Basis Generators are *Overloaded functions*.

harmonic_coordinate(double r, WaveFunction wf)
harmonic_coordinate(double r, int n, int l, double b)
Arguments:
r: Point to evaluate the Harmonic Oscillator wavefunction at
b: Length parameter
n: n quantum number of the wavefunction
l: l quantum number of the wavefunction
Return: (double) Evaluates R_{nl}(r)
Usage: The original basis for the many body problem, needed for inner-products in coordiante eigenspace

harmonic_momentum(double k, WaveFunction wf)
harmonic_momentum(double k, int n, int l, double b)
Arguments:
k: Momentum to evaluate the Harmonic Oscillator wavefunction at
b: Length parameter
n: n quantum number of the wavefunction
l: l quantum number of the wavefunction
Return: (double) Evaluates $\tilde{R}_{nl}$(k)
Usage: The original basis for the many body problem, needed for inner-products in momentum eigenspace

cs_coordinate(double r, WaveFunction wf)
cs_coordinate(double r, int n, int l, double b)
Arguments:
r: Point to evaluate the Radial wavefunction at
b: Length parameter
n: n quantum number of the Radial wavefunction
l: l quantum number of the Radial wavefunction
Return: (double) Evaluates S_{nl}(r)
Usage: Needed to evaluate inner-products

cs_momentum(double k, WaveFunction wf)
cs_momentum(double k, int n, int l, double b)
Arguments:
k: Momentum to evaluate the Radial wavefunction at
b: Length parameter
n: n quantum number of the radial wavefunction
l: l quantum number of the Radial wavefunction

Return: (double) Evaluates $\tilde{S}_{nl}$(k)
Usage: Needed to evaluate inner-products

3.5 Main

write_file(double x[ ], double y[ ] ,int n)
Arguments:
x: Array of Doubles, x-axis of data points
y: Array of Doubles, y-axis of data points
n: number of points in each array
Return: Void
Usage: Writes a text file (err.txt) in the format x0:y0,x1:y1,... Use spline_plot.py to plot this data.

4 Testing

The general test is to test the convergence of the integration as a function of the number of data points. Which tells roughly how many data points we will need to be accurate.

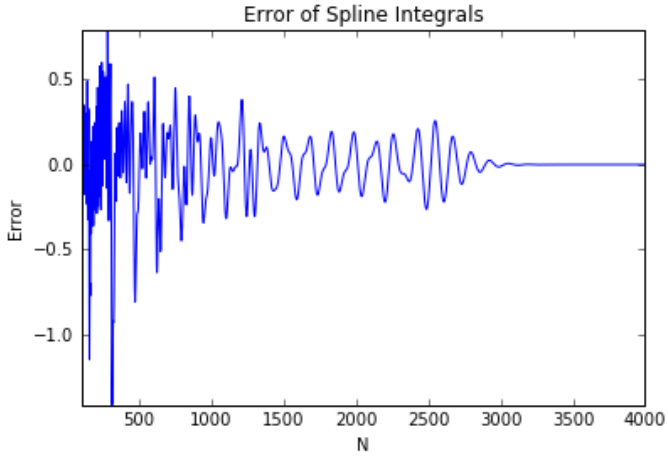

Figure 2: Convergence of $\int_0^\infty S_{50\,32}(r)S_{47\,32}(r)\,dr$ as a Function of Data Points

4.1 Mathematica Vs. CSI

Matematica was the standard I held random tests too, tests that I could not analytically solve myself. One of these tests being the evaluation of

$$\int_0^\infty sin^4\left(e^{-r}\right)dr \tag{4}$$

both Mathematica and CSI agreed that the integral was approximately 0.16063.

4.2 Testing each Routine

All polynomials were rigorously tested against Mathematica. It was found that Laguerre_Polynomials broke down for large values of x. This required me to use the GSL routines for Associated Laguerre functions.

The Laguerre basis functions were tested against

$$\int_0^\infty R_{n'l}(r)R_{nl}(r)\,dr = \delta_{n'n} \tag{5}$$

$$\int_0^\infty \tilde{R}_{n'l}(k)\tilde{R}_{nl}(k)\,dk = \delta_{n'n} \tag{6}$$

$$\int_0^\infty S_{n'l}(r)S_{nl}(r)\,dr = \delta_{n'n} \tag{7}$$

$$\int_0^\infty \tilde{S}_{n'l}(k)\tilde{S}_{nl}(k)\,dk = \delta_{n'n} \tag{8}$$

with $n, l < 20$, 1000 data points were sufficient enough to get an accurate result. However when $n, l > 40$, The data points needed to be scaled up to a maximum of 10000 until the change of variable broke down from discontinuities.

4.3 Debugging

If GSL takes an input it does not like it will throw its generic error back, sometimes leaving you baffled. The best process I found was explicitly checking the inputs of each GSL routine. For example making sure the number of points lined up with the size of the array. The only problem I am still not sure why it throws an error is in gsl_spline_eval_integ() it will not accept the range 0 to 1 but rather 0 to x[n-1] which means 0 to $1 - \frac{1}{n}$. Other common errors we thrown because of coding the function to evaluate was wrong or had a discontinuity I did not see. Simply use if statements to change the value to zero or evaluate the limit yourself.

5 Limits of CSI and Change of Variable

The GSL's cubic spline integration is great but it will break down eventually. The problems that always popped when CSI or change of variable failed were almost exclusively linked to discontinuities.

5.1 Dividing by Zero

When coding your function within the domain zero to one, you should make sure you are not evaluating 1/0. This may throw an error in GSL or cause your integral to be slightly off and in most cases will simply return nan as the value of your integral.

5.2 Slowly Converging Functions

When a function does not converge fast enough it will push all of the evaluated points into a very small range near 1, generating a vertical asymptote.

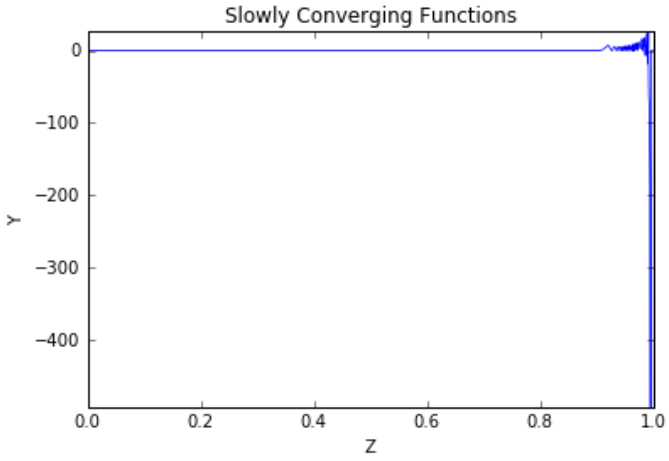

Figure 3: Evaluating $S_{60\,45}(z)S_{59\,45}(z)\,\frac{1}{(1-z)^2}dz = y$

Change of Variable breaks down as the function forms a vertical asymptote. Eventually the plot will just be zero all the way to one because all the information of the function was lost by the computer.

6 Resources

6.1 Change of Variable

Computational Physics
Chaper 5.8: Integrals over infinite ranges

6.2 Spline Interpolation

Spline Interpolation
-General overview of splines

Cubic Splines
-Specific detials on solving cubic splines

Numerical Recipes
-Coding Splines
Chapter 3: Splines
Section 3.3: Cubic Splines

GSL MANUAL
Chapter 28: Interpolation
Chapter 40: Splines
Chapter 7.22: Laguerre Functions

6.3 Polynomials

Laguerre
Jacobi

6.4 Laguerre Basis

Coulomb-Sturmian basis for the nuclear many-body problem
-Dr. Caprios paper: Defines R_{nl}, $\tilde{R}_{nl}$, S_{nl} *and* $\tilde{S}_{nl}$

6.5 Gauss Quadrature

Calculating Gauss Quadrature Rules
Section 2: How to find the roots of orthogonal polynomials

6.6 Additional C++ Information

Function pointers
Overloaded functions

7 Additional Comments

7.1 Future Updates

-I am working on recoding the polynomials using backwards reccurance for greatly improved accuracy.

7.2 Compiling

Process is the same as always. Build the object files, then create your executable by linking them.
g++ -std=c++11 -c example.cpp -I"GSL Header File Location"-Wall
g++ -std=c++11 -static example.o linkedfiles.o -L"GSL LIBs Location" -lgsl -lgslcblas -lm -o Name
You can always modify the make files yourself, simply modify the directories of Eigen,GSL libs and GSL headers.

7.3 Contact Information

If you have any questions, comments or find any errors please email me at floresab@msu.edu